Lec : 9

* 2nd Part : Search Techniques

→ Here, We will study how the Agents can make a decision According to the plan they put before.

→ On this lecture we will study 3 subjects.

- Agents that Plan Ahead
- Search Problems
- Uniformed Search Methods.

(A) Agents that Plan.

→ There are 2 types of Agents classified according to their work and interests

- Agents interest of complete the process (simple system)
- Agents interest of what will happen after ending the process to Find the best solution (complex system)

[1] Reflex Agents:

- Choose action based on current percept (and maybe memory)
  (Look_up_Table Contains of IF_Then Statement)
- May have memory or a model of the world's current state.
  (Interest of Current State)
- Do not consider the Future Consequence of their actions.
  (Do not Ask "what IF" & Do not interest of the result of the Actron it will end)
- Consider how the World IS
  (can't Care about the Future Actions)
- Can a reflex agent be retional ?
  Yes, IF we provided it by more IF conditions

ex: IF You Find An Apple and be sure that You wont Fall, Pick the apple

[2] Planning Agents.

- Ask "what IF"
- Decisions based on (hypothesized) consequences of actions.
- Must have a model of how the world envolves in response to actions (what will happen after ending the action.)
- Must formulate a goal (test) (must reach the goal)
- Consider how the world WOULD BE (LOOK For Good Future)
- Optimal vs Complete planning.

This Agents not only reach the goal or take an action but also it want to get the optimal solution of the problem (Search for the best way to get the goal OR the minimum Time can be token to reach the goal).

- Planning vs & Replanning

planning: Agents make a plan according to search so it take a period of time to plan then take a decision according to plan and start working.

. Plan only one time.

Replanning: Agents make a plan for the First step according to search then take a decision to complete First step After ending First step, it will replan For the next step according to new search and take a new best decision to start working for the next step

Here, For each step, Agents search, plan, make a decis an start working.

. Plan For each step ⇒ number of plans = number of Ste

→ Reflex Agents ⇒ Simple System

Planning Agents ⇒ Complex system

. Agents are classified as a reflex or a planning according to its work

Note: One Agent can be designed with Simple System to be reflex and the same agent can also be designed with Complex System to be Planning Agent 61

(B) Search Problems.

Search Problem: process to take right decision.

- A search Problem consist of :
  - A State Space (world)

    All the posible Cases of the Model (search Problem)

    ex: A mouse wants to eat all 9 piece of cheese

     Here, Mouse is eating the piece on the Center

     Here, The Mouse ate one piece and now is eating another one.

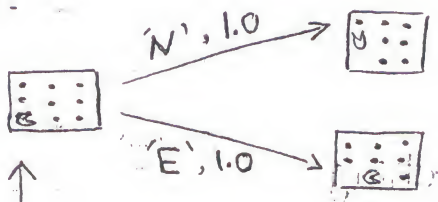
     Here, The Mouse eating the Last piece

    ...

    .There are a big number of cases that can describe the world

  - A successor Function (actions, Costs)
    - For any State, what is the action that I can take and what is the cost of taking it

    ex:

    For this case we can take one of 2 decisions (Actions)

    1. Go North with 1.0 cost
    2. Go East with 1.0 cost

  - A start State and a goal test

    Start State: State From which I can start working

    Goal test: The Final State

(B) Search Problems.

Search Problem: process to take right decision.

- A search Problem consist of:
  - A State Space (World)

    All the posible cases of the Model (search Problem)

    ex: A mouse wants to eat all 9 piece of cheese

     Here, Molise is eating the piece on the Center

     Here, The Mouse ate one piece and now is eating another one.

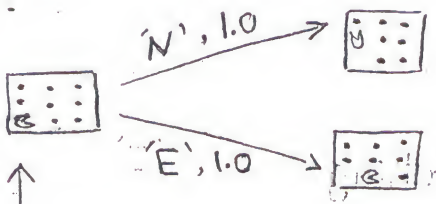
     Here, The Mouse eating the Last piece

    ...

    .There are a big number of cases that can describe the world

  - A successor Function (actions, Costs)
    - For any State, what is the action that I can take and what is the cost of taking it

    ex:

    For this case we can take one of 2 decisions (Actions)

    1. Go North with 1.0 Cost
    2. Go East with 1.0 Cost

  - A start State and a goal test

    Start State: State From which I can start working

    goal test: The Final State

- A solution of search Problem: is a sequence of actions (a plan) which transforms the start state to a goal state.

Example: Traveling In Romania

Show map at Slide 4 lec 3

- State Space:

  All Cities

- Successor Function:

  Roads: Go to adjacent City with Cost = distance.

- Start State & goal Test

  Start state: Arad

  goal Test : Bucharest

- Solution

  . There are 8 ways From Arad to Bucharest

  → we want to find optimal way with smallest distance.

* —— *

→ The World State includes every last detail of the environment

→ A search state Keeps only the details needed for Planning (abstra

we studied 2 types of search Problems, we will compare between the

Problem	Pathing	Eat All Dots
States	(x, y) Location	(X, y), dot booleans
Actions	Moves (North, East, South, West)	Moves (North, South, East, West)
Successor	update Location only	update Location and possibly a dot boolea
Goal Test	is (X, y) = END	Dots all False.

* State Space Sizes ?

Example : Paeman Game

→ Content : 30 dots, Agent, 2 Ghost الشبح

→ Goal : Agent must eat all dots without being Kild by any Ghost

. World state :

- Agent Positions : 120
- Food Count : 30
- Ghost Positions : 12
- Agent Facing : North, South, East, Wast

. How Many ?

- World states ? (all possible states)

  $120 \times (2^{30}) \times (12^2) \times 4$ ⇒ Very big Number

- States For Pathing ?

  120 (all Agent possible positions)

- States For all dots ?

  $120 \times (2^{30})$

→ From this Example, we Note that, World State is too big and this will make the system more Complex. So, we will use Search State in which we can take only needed states like states For pathing IF we are more interesting about pathing OR States For all dots IF eating all dots is our goal.

In this case our chosen world state will be Simple.

For The Same Example "Pacman Game"

→ IF we add anew Condition what about the State Space?

- problem: eat all dots will Keeping the ghosts perma-scared
  → New Condition: Don't Appear ghosts for a period of time at the start of the game.

Select all Components that are required in the state-Space representation for the search problem:

- You may assume that the ghosts are already scared at the start of the game.
- Search State:
  1. Pacman's position
  2. The position of each ghost
  3. A boolean for each dot that represents whether it has been eaten.
  4. A boolean for each power pellet that represents whether it has been eaten
  5. The remaining amount of time for which the ghost will be scared all previous locations that each ghost has visited

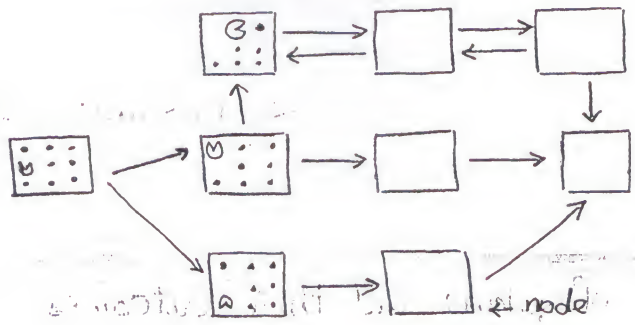
- What does the state Space have to Specify?
  → Agent position
  → Dot booleans
  → power pellet booleans
  → Remaining Scared Time.

© Uniformed Search Methods

(there is no information about the goal Location or how far it is from us)

① State Space Graph

A mathematical representation of a search Problem



- Nodes are (abstracted) world Configurations
- Arcs represent seccessors (action results).
- The goal test is a set of goal nodes (may be only one)

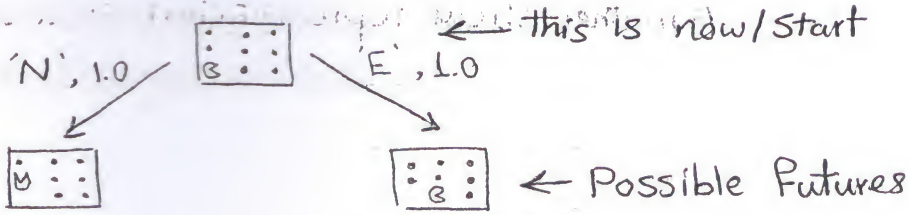
* In a search graph, each state occurs only once.
  - → we make State Space graph so we represent needed states only, as a result each state appears one time.
    Here, we need to reach a solution not the optimal solution
  - → we make world state graph to get optimal solution but this need more memory size.

* We can rarely build this Full graph (world state graph) in memory (it's too big), but it's a useful idea.

* In a search graph, there is no start state but a goal test.

[2] Search Trees

→ Can be considered as a part of Search graph



- A search tree:
  - A "what if" tree of plans and their outcomes.
  - The Start State is the root node
  - Children correspond to successors
  - Nodes show states, but correspond to PLANS that achieve those states
  - For most problems, we can never actually build the whole tree

* State Graphs & Search Trees.

* For State graph, You need to represent all Configurations but For search tree, You start from the root and represent configurations step by step until reaching to goal so you don't need to represent all configuration. Note: You need to represent all configuration using search tree If You search For optimal Solution

* Each NODE in the search tree is an entire PATH in problem graph.

* Search Tree has a start state and goal Test but search graph has only goal Test

* Searching With a Search Tree

- Try To expand as few tree nodes as possible.
- Maintain a fringe of partial plans under consideration
- Try to expand as few tree nodes as possible.

→ General Tree Search

Function TREE-SEARCH (Problem, Strategy) return a solution, or failure initialize the search tree using the initial state of problem

Loop do

If there are no candidates for expansion then return failure

Choose a leaf node for expansion according to Strategy

If the node contains a goal state then return the corresponding solution.

else expand the node and add the resulting nodes to the search tree

end

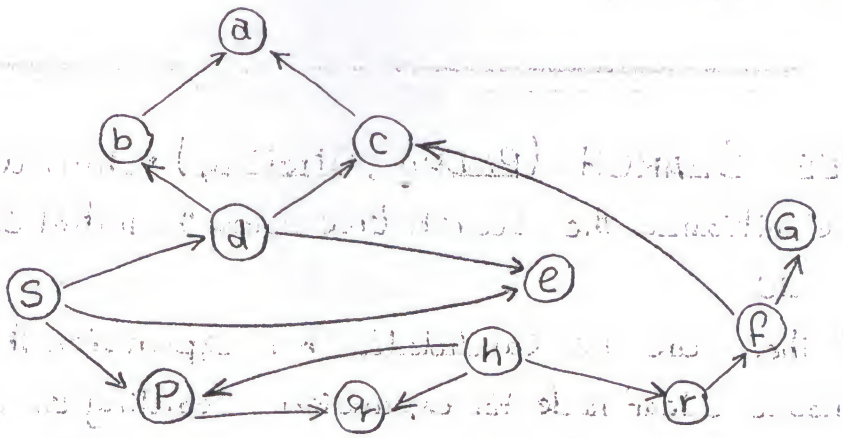
* ——— *

* Important Ideas

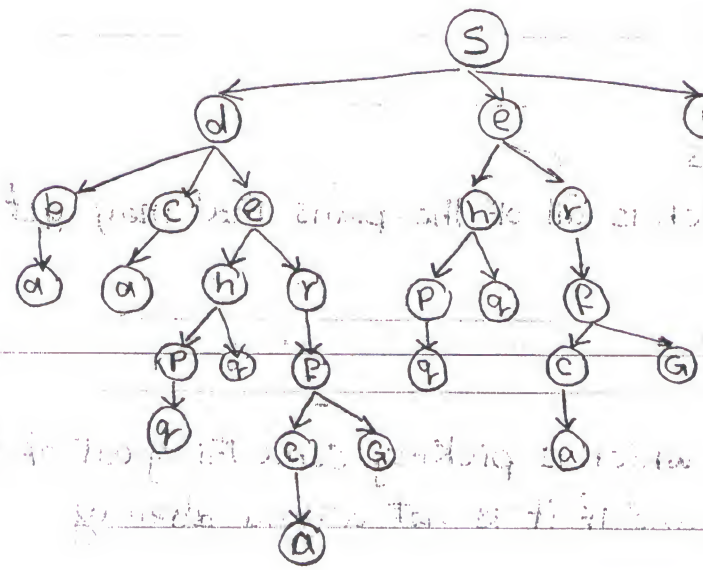
- Fringe : which is all of the plans that may yet work. all ways that can reach me from start to goal during world state search tree. So one of this Frings is my solution (optimal solution)
- Expansion : which is picking some thing out of the fringe and if it is not a goal already. process of expanding new state from the current state on one of Fringes. we can't do this process if we reach to the goal.

- Exploration Strategy: what fringe nodes do you explore next?
  - what will happen next?
- Main question
  - Which fringe nodes to explore? to reach to goal.

Example



Search Tree



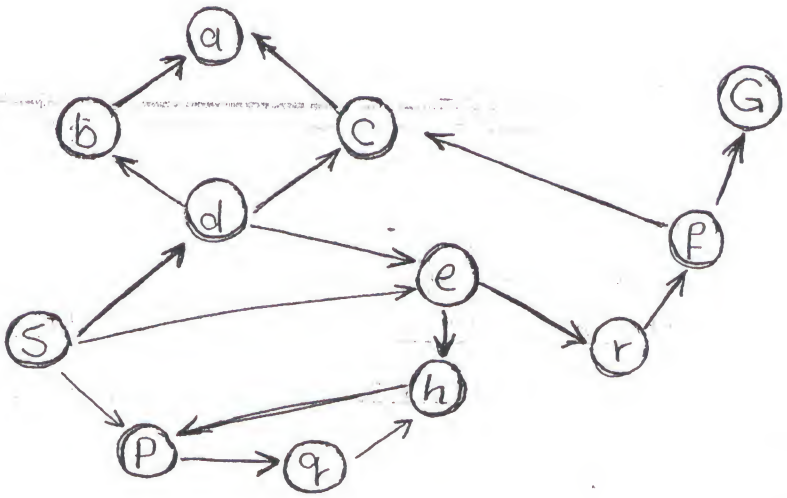
Fringes: S → d → e → r → f → G

S → e → r → f → G

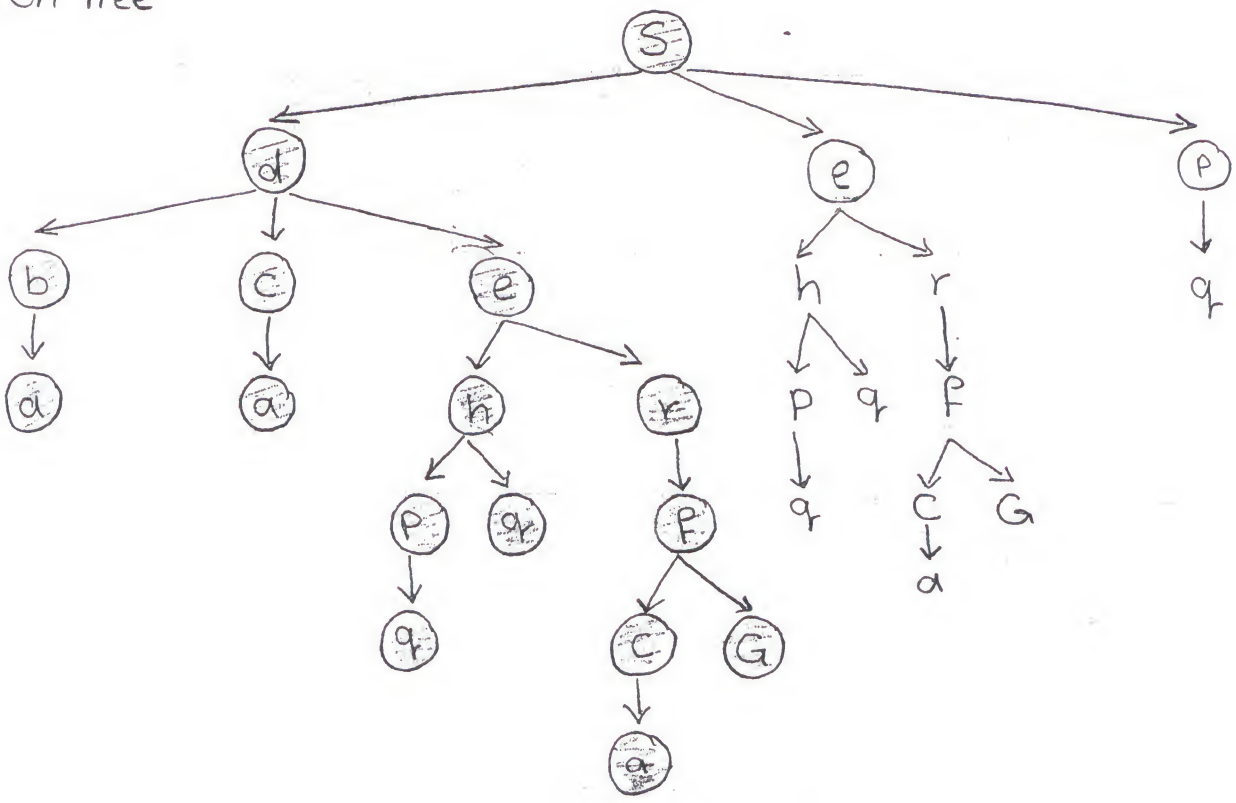
1 Depth-First Search.

Strategy: expand a deepest node First
Implementation: Fringe is LIFO stack.
Solution: Left most solution
Example:

Search Graph



Search Tree



Implementation : Fring is a LIFO stack.

[1] Push root in stack

[2] From S we can take d or e or p
Then we take left mast child to be pushed in stack according to search Tree so we push d

[3] From d we can take b or c or e to be pushed in stack ; We will take left mast
we push b

[4] From b we push its only child a

[5] we reach to last child but not the Goal
So, Pope a, b (pope until reach to parent with unvisited childrens)

[6] push next child of d ⇒ C

[7] push a then we don't reach to goal so pope a, C

[8] push e → push h → push p → push q (not goal)
pope q → pope p
push q (not goal) → pope q, h
push r → push f → push → C → push a
(not goal) pope a, C
push G (the goal)

∴ The solution is S → d → e → r → f → G

Stack final Content

* Search Algorithm properties

For Any Algorithm we check For 4 properties.

1. Complete: Ability to reach to a goal if it is exist
   → For any Finite Search Algorithm we can find a goal if it is exist and any algorithm we use will be Complete algorithm but For any infinite Search Algorithm we may not find a goal although it is exist because we may enter infinite loop

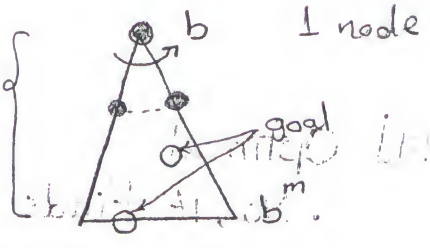
2. Optimal: Ability to reach to Best Solution

3. Time Complexity: Time to reach to a Solution.
   Time to expand all nodes of one fringe equal to Time to expand one node of on fringe X number of fringe nodes.

4. Space Complexity: Size of Fringe in memory
   Size of nodes of fringe in stack.

* Cartoon of Search tree:

- b is the branching Factor
- m is the maximum depth
- Solutions at Various depths

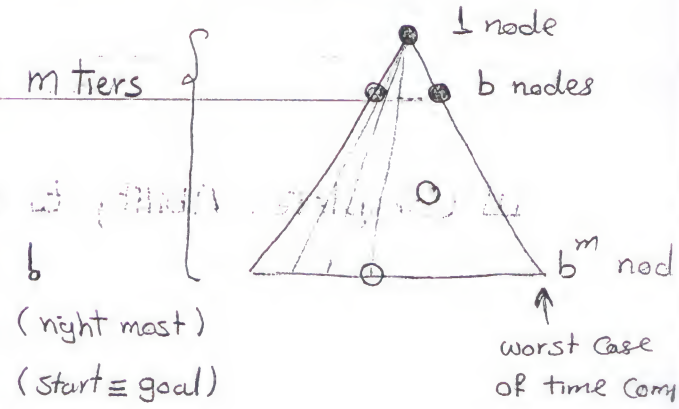


* Number of nodes in entire tree

$$1 + b + b^2 + \cdots + b^m = O(b^m)$$

→ branching Factor: Start node can make a branch for b number of children

* Depth-First Search (DFS) Properties



1 Time Complexity

what nodes DFS expand ?

- Some left prefix of the tree
- Could process the whole tree !

worst Case expand all nodes (right most)

best Case expand no node (start ≡ goal)

general → • If m is finite, take time $O(b^m)$

Time of worst Case = $O(b^m)$

Time of best Case = $O(1)$

2 Space Complexity

- Only has Siblings on path to root, so $O(b^m)$

size of Fringe at Stack (memory)

worst Case we get solution on the last tier

3 Complete

- m Could be infinite, so only if we prevent Cycles

Solution Can be found only if

1. it is exist 2. finite Algorithm (start and end are Known)

→ worst Case we have infinite number of nodes, so it won't be Complete.

4 Optimal

- No, it Finds the "leftmost" Solution, regardless of depth or Cost.

→ in this algorithm if we find left most solution, we won't Complete Tree so we don't get Optimal Solution

→ If we get all solutions we can't choose the best one. because there are many problems like ununary depth or Cost

* 2 * Breath - First Search

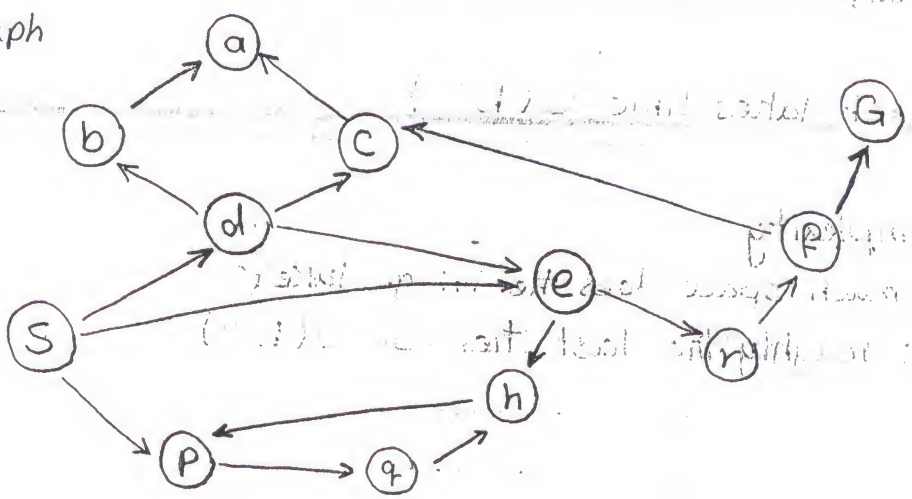
Strategy: expand a shallowest node First

Implementation: Fringe is a FIFO queue

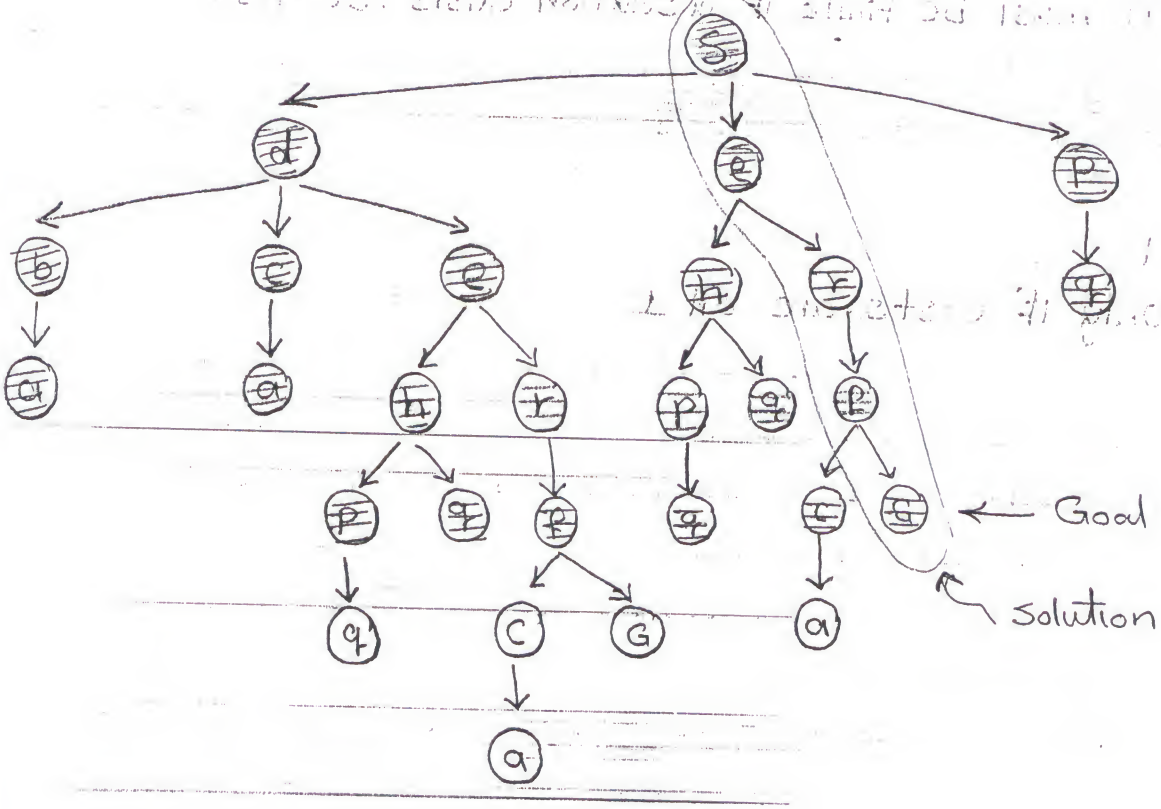
Solution: Shallowest Solution (here we get shortest fringe)

Example:

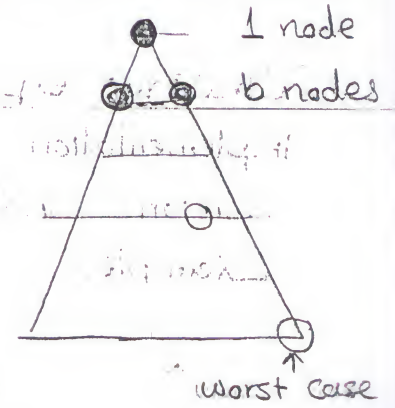
Search Graph



Search Tree.



* Breadth-First Search (BFS) Properties



[1] Time Complexity

what nodes does BFS expand?

- processes all nodes above shallowest solution
- let depth shallowest solution be m
  worst case in which we expand all nodes

general → • Search takes time $O(b^m)$ (worst case)

[2] Space Complexity

How much space does the fringe take?

- Has roughly the last tier. So $O(b^m)$

→ size of fringe in memory.

general : at the worst case the solution is on the last tier

[3] Complete:

- m must be finite if a solution exists, so Yes!

→ If a solution is exists we can find it before entering infinite loop So it is Complete Forever.

[4] Optimal

- Only if Costs are all 1

→ We get shallowest solution so if Costs equal to 1, we get optimal solution

→ If Costs don't equal to 1, we can't be sure that we get optimal solution

→ here we can't say optimal solution (less Cost) is the shallowest solution.

→ general it is not optimal Algorithm

* DFS & BFS

- When will BFS outperform DFS ?

  BFS will outperform DFS when

  1. Need less Time Complexity
  2. Need Shallowest Solution
  3. Need Optimal Solution For Costs equal to 1
  4. Need Complete Solutions.

- When will DFS outperform BFS ?

  DFS will outperform BFS when

  1. most Solutions at left side of tree (given)
  2. all solutions at the last level (given)
  3. Need less Space Complexity

* 3 * Iterative Deepening

- Idea: Get DFS's space advantage with BFS's time / Shallow Solution advantages (Collection of DFS and BFS).
  - Run a DFS with depth limit 1, IF no solution
  - Run a DFS with depth limit 2 , IF no Solution
  - Run a DFS with depth limit 3 , ...

  * at BFS we expand all nodes level by level this mean that in the level that contain solution we expand all nodes whever the solution can be on the leftmost of the tree

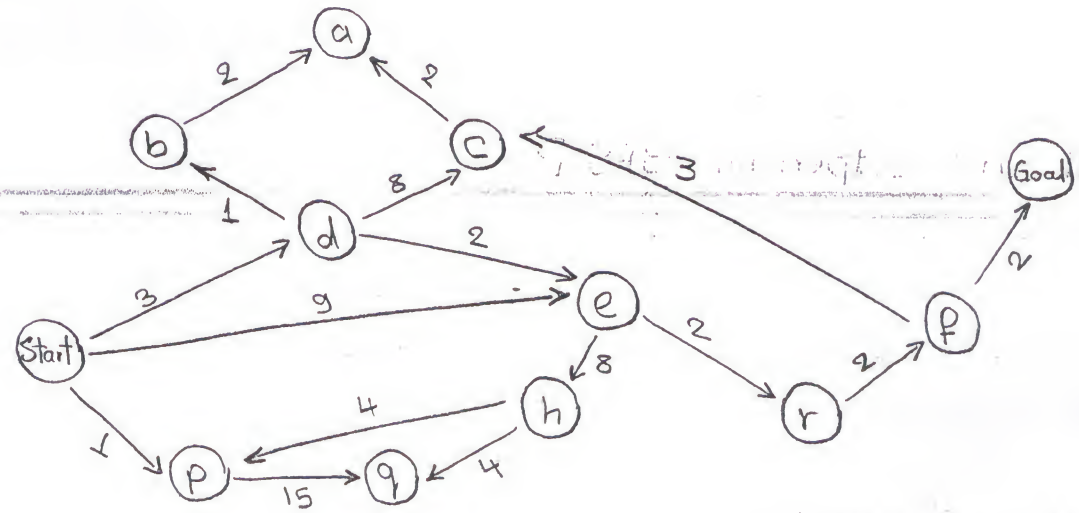
---

→ Here, In Iterative Deepening we expand level by level For one level we expand from leftmost to reach to goal When we Find goal we stop expanding So, here we get Shallowest Solution by the Consept of DFS

* Here we Improve Time Complexity (BFS disadvantage) and Size Complexity (DFS disadvantage)

- Isn't that wastefully redundant?
  - Generally most work happens in the lowest level searched, so not so bad!

* Cost-Sensitive Search.



- BFS finds the shortest path in terms of number of actions.
- It does not find the least-cost path.
- We will now cover a similar algorithm which does find the least-cost

* To expand any node there is a cost depending on the search problem
  (problem: pathing ⇒ cost: length of distance between every 2 cities)
* In some search problems, the cost of expanding any node is equal
* In another search problems, every expansion has a different cost

* 4 * Uniform Cost Search.

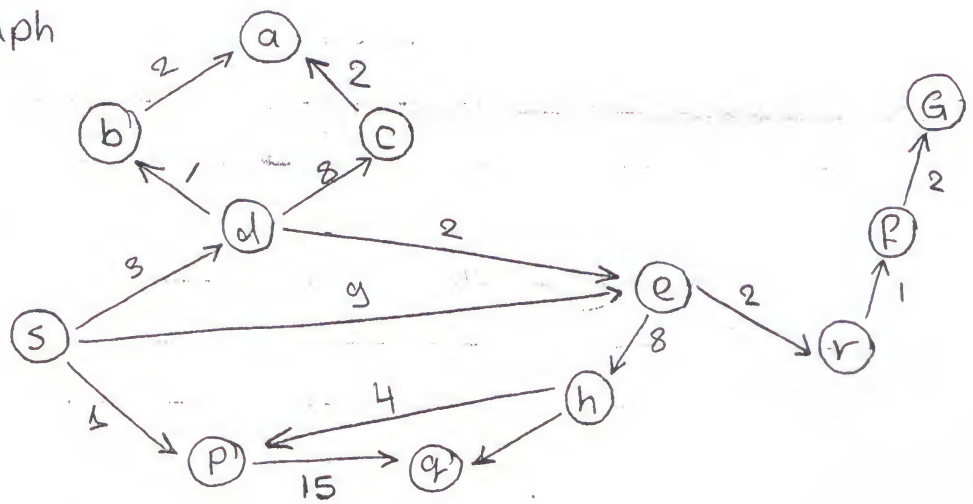
Strategy: expand a cheapest node first

Implementation: Fringe is a priority queue. (Priority: Cumulative (تراکمی) Cost)

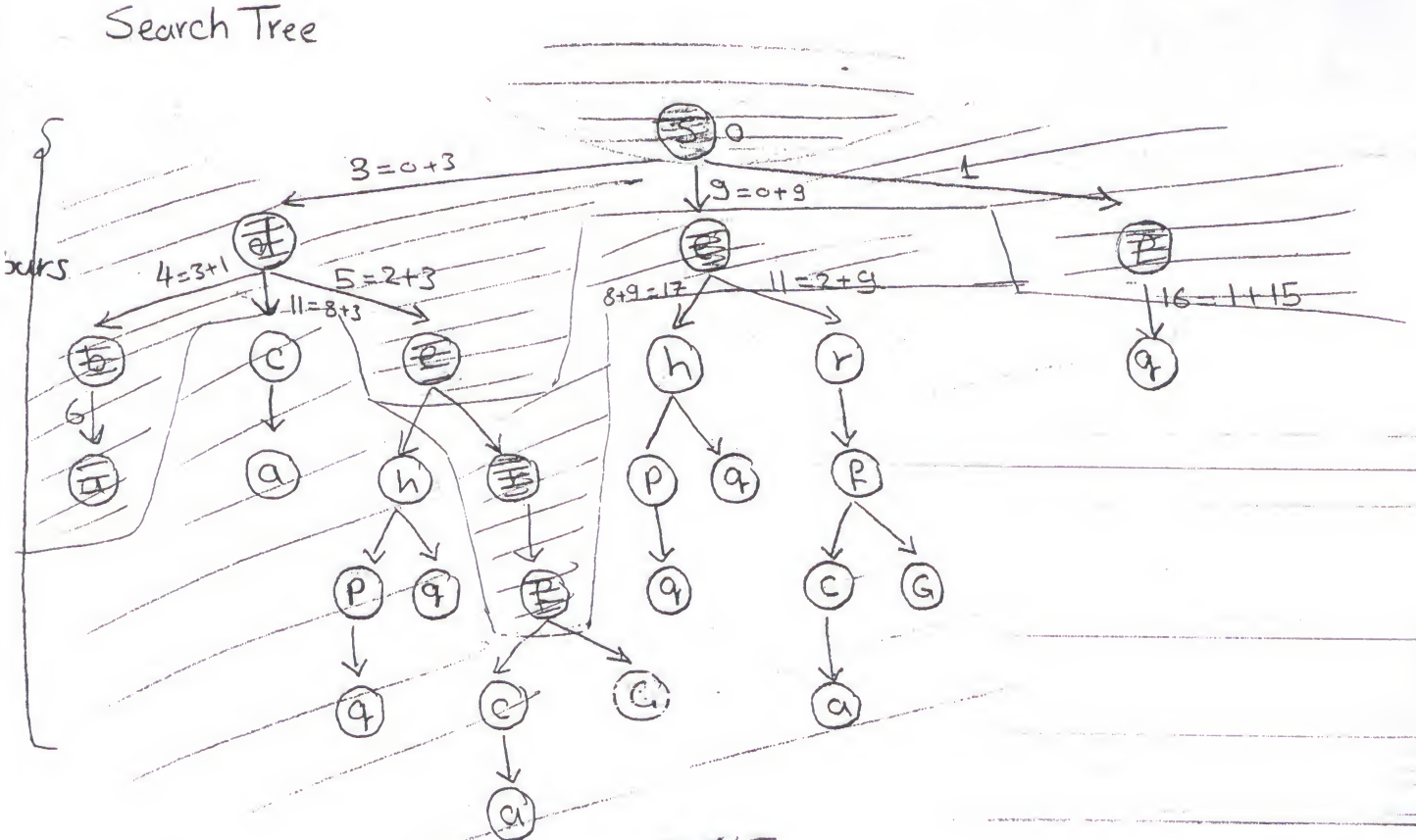
Solution: Cheapest Solution !

Example:

Search graph



Search Tree



V.I → Note: To get the Cost of expanding any node, Calculate The Cost from the start node to this node.

- Using Uniform Cost Search, we expand the node of the cheapest cost

⇒ For The Last Example

1. expand node S of 0 Cost
2. we have 3 nodes can be expanded d, e, p (Children of S)
   Cost of d = 0+3 = 3 Cost of e = 0+9 = 9
   Cost of p = 0+1 = 1 So, we expand p
3. we should expand child of p ⇒ q
   Cost of q = 0+1+15 = 16 (High Cost), So we go to expand d
4. we should expand child of d ⇒ b, c, e
   Cost of b = 0+3+1 = 4 Cost of c = 0+3+8 = 11
   Cost of e = 0+3+2 = 5 ⇒ we expand b
5. we should expand child of b ⇒ a
   Cost of a = 0+3+1+2 = 6 (High Cost), So we go to expand e
6. we should expand child of e ⇒ r (High Cost), go to expand c
   (we reach to last node but not the goal), go to expand r
7. we should expand child of r ⇒ f Cost of f = 0+3+2+2+1 =
8. we should expand child of f ⇒ G Cost of G = 0+3+2+2+1+2=
   G High Cost, go to expand e
9. we should expand child of e ⇒ h, r (2 are high Cost)
   go to expand G (goal)

* Uniform Cost Issues

Advantages

1. UCS is complete and optimal

Disadvantages

1. No information about goal location

   in the last example we reach to goal, but don't expand it because there is another node more cheap we expand it at first and then expand goal node.

   Solution to make an algorithm give us information about how far we are from goal (this solution is in another search algorithms)

2. explores options in every "direction"

   Idea of contours that we expand nodes in all directions

   → There is no determine direction for our work.

* The One Queue : Priority Queues

- All these search algorithms are the same except for fringe strategie

  we can use queue for All Algorithms

  — Conceptually, all fringes are priority queues.
  (i.e collections of nodes with attched priorities)

  — practically, For DFS and BFS, you can avoid the log(n) overhead from an actual priority queue with stacks and queues.

  — Can even code one implementation that takes a variable queuing object.

  → using priority queue (priority for deepest, shallowest ..) For any algorithm but this will cause more time complexity

  For DFS and BFS we aren't in need of using it as we don't interest of priority. but it is important to use it in UCS as it interest of cost priority.

* Uniform Cost Search (UCS) Properties.

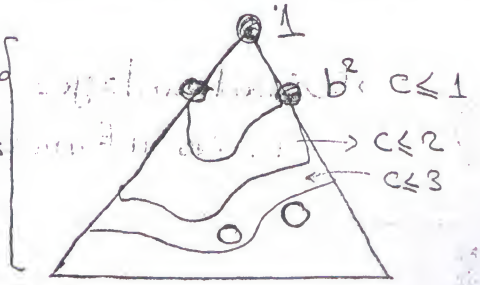
[1] Time Complexity

What nodes does UCS expand?

- processes all nodes with cost less than cheapest solution!

Worst case solution on the last contour

- IF that solution costs C^* and arcs cost at least ε, then C^*/ε tiers



'effective depth' is roughly C^*/ε

we need to know cost of contours that can be expanded

C^* : Total cost of all contours

ε : number of expanded contours

$\therefore$ Cost of expanded contours $= \frac{\text{Total Cost}}{\text{N. of Contours}} = \frac{C^*}{\varepsilon}$

To get number of expanded nodes $= (b^{C^*/\varepsilon}) \Rightarrow$ For the worst case we expand all nodes

- Take time $O(b^{C^*/\varepsilon})$ (exponential in effective depth).

[2] Space Complexity

How much space does the fringe take?

- Has roughly the last tier, so, $O(b^{C^*/\varepsilon})$

Generally : worst case $\Rightarrow$ Solution found on the last contour

[3] Complete

- Assuming best solution has a finite cost and minimum arc cost is positive $\Rightarrow$ Yes!

IF the solution exists, we find it for infinite or finite algorithms

[4] Optimal

- Yes! (Proof next lecture via A*)

optimal as we find cheapest solution.

* Search Operates over model of the world.

- The agent doesn't actually try all the plans out in the real world!
- planning is all "in simulation"
- Your search is only as good as your models

* Time Complexity
  1. DFS (more Time complexity
  2. BFS
  3. UCS (less Time complexity

* Space Complexity
  1. DFS (less space complexity)
  2. BFS (more space complexity)
  3. UCS (space complexity)

* Complete
  1. DFS (not complete generally)
  2. BFS (Complete)
  3. UCS (Complete) ✓

* Optimal
  1. DFS (not optimal)
  2. BFS (not optimal generally)
  3. UCS (Optimal) ✓